DAM
Desarrollo de Aplicaciones Multiplataforma
2º Curso

AD
Acceso a Datos

UD 5
Bases de Datos Orientadas a Objetos (BDOO)

IES BALMIS
Dpto Informática
Curso 2019-2020
Versión 1 (03/2019)

UD5 – Bases de Datos Orientadas a Objetos

ÍNDICE

1. Qué son las bases de datos orientadas a objetos

Una **base de datos orientada a objetos** (o, de forma más detallada, un "Sistema de Gestión de Bases de Datos Orientado a Objetos", SGBDOO, en inglés "Object-Oriented Database Management System", OODBMS) es un sistema de gestión de base de datos en la que se representa la información en forma de objetos, tal y como se utilizan en la programación orientada a objetos, al contrario que en las bases de datos relacionales, que están orientadas a tablas, o las objeto-relacionales, que son un híbrido entre ambos enfoques.

Algunas bases de datos orientadas a objetos están diseñados para integrarse bien con lenguajes de programación orientados a objetos tales como Java, C#, Python o Ruby (en ocasiones, exclusivamente con uno de ellos); otras, como JADE tienen sus propios lenguajes de programación. En cualquier caso, los SGBDOO usan exactamente el mismo modelo que los lenguajes de programación orientados a objetos.

2. Historia de las bases de datos orientadas a objetos

El término "Sistema de Base de Datos Orientada a Objetos" apareció por primera vez cerca de 1985.

Los primeros productos comerciales eran herramientas como GemStone, GBASE y Vbase. Durante la primera mitad de la década de 1990 llegaron otros como Jasmine (de Fujitsu, comercializado por Computer Associates), Matisse y O2 (adquirida por Informix, que posteriormente fue a su vez adquirida por IBM).

Matisse sigue existiéndose, y tiene una versión gratuita (pero que requiere registro) en su web matisse.com, la que llaman "Developer's version".

Los SGBDOO añadían el concepto de persistencia a los lenguajes de programación orientados a objetos. De hecho, los primeros productos comerciales se integraban con ciertos lenguajes concretos, por ejemplo GemStone con Smalltalk y GBASE con LISP.

Durante gran parte de los años noventa, C++ dominó el mercado comercial de los SGBDOO. A finales de los 90 se comenzó a añadir **Java** y, más recientemente, C#.

A partir de 2004, las bases de datos orientadas a objetos han visto un segundo periodo de crecimiento, con la aparición de gestores de código abierto accesibles y fáciles de usar. Es el caso de **DB4o** de Versant (db4objects), para Java.

Se puede descargar la última versión desde https://sourceforge.net/projects/db4o/

3. Características técnicas

La mayoría de las bases de datos orientadas a objetos también ofrecen algún tipo de lenguaje de consulta, permitiendo buscar objetos con algún lenguaje de programación declarativa, parcialmente similar a SQL. La sintaxis de cada lenguaje empleado, la forma de integrar las consultas con el lenguaje base y la interfaz de navegación de los datos son los detalles que marcan las mayores diferencias entre unos productos y otros.

El ODMG (Object Data Management Group) creó 4 revisiones de una especificación, con la intención de estandarizar todos estos detalles. La última versión de la especificación es la 3.0, de 2001, tras la que se disolvió el grupo. Esta especificación incluye, entre otros detalles: un modelo de objetos, un lenguaje de especificación de objetos (ODL, Object Definition Language), un lenguaje de consulta (OQL, Object Query Language, que toma de SQL muchas de sus ideas básicas) y la especificación de cómo conectar desde los lenguajes C++, Java y Smalltalk.

En una base de datos orientada a objetos, el acceso a los datos puede llegar a ser más rápido que en una relacional basada en tablas, ya que en general las operaciones tipo "join" no suelen ser necesarias, sino que los objetos se pueden recuperar directamente, siguiendo una serie de punteros enlazados.

Algunas bases de datos orientadas a objetos, por ejemplo GemStone o VOSS, ofrecen soporte para el control de versiones. Por una parte, un objeto se puede considerar el conjunto de todas sus versiones. Pero, por otra parte, también se pueden emplear de forma independiente las versiones de cada objeto.

Además, muchas bases de datos orientadas a objetos también permiten utilizar otras características habituales en bases de datos relacionales recientes, como la definición de restricciones ("contraints") y la creación de disparadores ("triggers") para que sean capaces de responder a eventos.

Instalación de Db4o

Aunque Db4o posee dos modos de trabajo: Modo Embebido y Modo Cliente/Servidor, nosotros usaremos el modo embebido para nuestras prácticas en desarrollo.

En un sistema ya en producción sería recomendable el Modo Cliente/Servidor ya que permitiría el uso de la BD por varios usuarios simultáneamente.

Para funcionar en el Modo Embebido solo es necesario utilizar la librería:

db4o-X.X.X-all-java5.jar **o** **db4o-X.X.X-java5.jar**

Las características de este servidor en modo embebido son:

SGBD	Db4o en modo Embebido
Modelo de Datos	Orientada a Objetos
Tipo de Datos	Objectos
Acceso	Fichero
Puerto	NO
Client Command-Line	NO
Client GUI Escritorio (Interfaz Gráfica de Usuario)	Object Manager
Client GUI Web (Interfaz Gráfica de Usuario)	NO

Características

- **Precio:** gratuito
- **Almacenamiento**: un único archivo

4. Un ejemplo real con Db4o

A diferencia el mapeo objeto-relacional de Hibernate, Db4o sigue un enfoque totalmente orientado a objetos, con sus ventajas (más fácil de integrar en un programa Java) y sus inconvenientes (no se puede usar SQL ni ningún lenguaje con una potencia similar).

Los pasos básicos para guardar datos serán poco más que usar "**.openFile**" para abrir el fichero (o crearlo la primera vez, si no existe), "**.store**" para preparar un dato que se añadirá al fichero y "**.commit**" para guardar los cambios. Finalmente, se deberá guardar el fichero con "**.close**".

Para abrir la Base de Datos en Modo Embebido, se utiliza una clase diferente según sea la versión 8.0 o la 7. Nosotros usaremos la 7 para poder usar el GUI Object Manager 7.4.

```
// Versión 8.0
db=Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),"./datos/personas.dat");

// Versión 7
db=Db4o.openFile("./datos/personas.dat");
```

Veamos un programa que desea guardar en Db4o objetos de la clase Persona:

```
package Clases;

public class Persona {
  private String nombre;
  private String apellidos;
  private int edad;

  public Persona() {
  }

  public Persona(String nombre, String apellidos, int edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }

  public String getNombre() {
    return nombre;
  }
  public void setNombre(String value) {
    nombre = value;
  }

  public String getApellidos() {
    return apellidos;
  }
  public void setApellidos(String value) {
    apellidos = value;
  }

  public int getEdad() {
    return edad;
  }
  public void setEdad(int value) {
    edad = value;
  }

  public String toString() {
    return "Persona: " + "nombre = " + nombre + ", " +
        "apellidos = " + apellidos + ", " + "edad = " + edad;
  }

  public boolean equals(Object otro) {
    if (otro == this) return true;
    if (!(otro instanceof Persona)) return false;
    Persona other = (Persona) otro;
    return (this.nombre.equals(other.nombre) &&
         this.apellidos.equals(other.apellidos) && this.edad == other.edad);
  }

}
```

Para poder grabar datos de objetos de la clase Persona, utilizaremos Db4o. Para que podamos compilar, es necesario añadir las clases de java de Db4o incluidas en el archivo **db4o**-99999999999-**all-java5.jar**

```
import Clases.Persona;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

public class EjemploDb4o1 {
  public static void main(String[] args) throws Exception {
    ObjectContainer db = null;
    try {
      db=Db4o.openFile("./datos/personas.db4o");
      Persona p1 = new Persona("Juan", "López", 22);
      Persona p2 = new Persona("José", "Álvarez", 25);
      db.store(p1);
      db.store(p2);
      db.commit();

      System.out.println("BD creada");
    } finally {
      if (db != null) db.close();
    }
  }
}
```

Como curiosidad, si lanzamos este programa una segunda vez, el fichero ya existe, de modo que no se crea un nuevo fichero ni se destruye su anterior contenido. De hecho, se vuelven a guardar dos datos (idénticos a los anteriores) al final de los ya existentes, igual que ocurriría en una base de datos relacional en la que no existiera clave primaria.

A la hora de leer datos desde un programa, usaríamos "**.queryByExample**" para buscar datos similares a uno dado. La lista resultante, se puede recorrer con "**while (lista.hasNext()) …**", obteniendo cada nuevo dato con "**lista.next()**", así:

```
import Clases.Persona;
import com.db4o.*;

public class EjemploDb4o2 {
  public static void main(String[] args) throws Exception {
    ObjectContainer db = null;
    try {
      db=Db4o.openFile("./datos/personas.db4o");

      // Añadimos otro "Juan"
      Persona p = new Persona("Juan", "Pérez", 21);
      db.store(p);
      db.commit();

      // Mostrar número de registros
      System.out.println("BD con "+
            db.queryByExample(new Persona(null, null, 0)).size()+" registros");

      // Buscamos todos los "Juan" (el campo con valor null o 0 no se filtra)
      ObjectSet<Persona> juanes=db.queryByExample(new Persona("Juan", null, 0));
      while (juanes.hasNext()) {
        System.out.println(juanes.next());
      }
    } finally {
      if (db != null) db.close();
    }
  }
}
```

Si hemos lanzado dos veces el primer programa y una vez el segundo, en el resultado aparecerán dos personas "Juan López" y un "Juan Pérez":

```
BD con 5 registros
Persona: nombre = Juan, apellidos = Lopez, edad = 22
Persona: nombre = Juan, apellidos = Lopez, edad = 22
Persona: nombre = Juan, apellidos = Perez, edad = 21
```

En el caso de "Db4o", el formato para esa consulta es crear un nuevo objeto en el que algún atributo (o varios) tenga(n) valor, y ese será el criterio de búsqueda. Los atributos que no se quieran usar como criterio se deberán dejar a "null", y a cero los valores numéricos. Esto deja entrever una limitación: no se podrán buscar valores cero, porque se interpretará como que no es un criterio de búsqueda sino un dato que no se desea tener en cuenta.

No existen claves primarias, sino identificadores de objeto (OIDs) a nivel interno de "Db4o", por lo que para evitar duplicados deberá ser el propio programador el que compruebe si ese objeto ya aparece en la base de datos, antes de intentar guardar un dato, como en este ejemplo:

```
import Clases.Persona;
import com.db4o.*;

public class EjemploDb4o3 {
  public static void main(String[] args) throws Exception {
    ObjectContainer db = null;
    try {
      db=Db4o.openFile("./datos/personas.db4o");

      // Añadimos otro "Juan"
      Persona p = new Persona("A", "B", 30);

      // Vamos a intentar guardar 10 veces el mismo dato
      for (int i=1; i<=10; i++) {
        if (db.queryByExample(p).hasNext() == false) {
          // Si no existe, guardamos
          db.store(p);
          db.commit();
        }
      }

      // Y finalmente buscamos todos los apellidos "B"
      ObjectSet<Persona> lista=db.queryByExample(new Persona(null, "B", 0));
      while (lista.hasNext()) {
        System.out.println(lista.next());
      }

    } finally {
      if (db != null) db.close();
    }
  }
}
```

En este caso, intentamos guardar 10 veces el mismo dato. Como estamos comprobando si ya está o no, solo se guardará la primera vez, pero no las siguientes.

Para **modificar** un dato, bastará con recibirlo desde la base de datos, cambiar lo que se necesite y finalmente volver a guardar usando el mismo objeto:

```
import Clases.Persona;
import com.db4o.*;

public class EjemploDb4o4 {
  public static void main(String[] args) throws Exception {
    ObjectContainer db = null;
    try {
      db=Db4o.openFile("./datos/personas.db4o");

      // Vamos a modificar la edad de la ficha de "A B"
      ObjectSet<Persona> listamod = db.queryByExample(new Persona("A", "B", 0));
      if (listamod .hasNext()) {
        Persona a = (Persona) listamod.next();
        a.setEdad(a.getEdad() + 1);
        db.store(a);
        db.commit();
      }

      // Y finalmente buscamos todos los apellidos "B"
      ObjectSet<Persona> lista=db.queryByExample(new Persona(null, "B", 0));
      while (lista.hasNext()) {
        System.out.println(lista.next());
      }

    } finally {
      if (db != null) db.close();
    }
  }
}
```

La forma de hacer **búsquedas más complejas** no es tan sencilla, porque QBE (QueryByExample) gestiona bien las búsquedas exactas, pero no las comparaciones (por ejemplo, apellidos que empiecen por un cierto texto o edades por encima de cierto valor).

Hay básicamente tres alternativas, según el gestor de bases de datos orientado a objetos que se emplee:

- **Analizar manualmente** todos los datos mediante un bucle, lo que puede resultar en algunos casos complicado de programar y además puede ser muy lento, especialmente cuando los datos se reciben a través de una red.
- Usar "**predicados**" (un predicado es básicamente una función que, a partir de ciertos parámetros, devuelve verdadero o falso) que contengan otros predicados, lo que provoca código mucho menos legible que el de QBE, como en este ejemplo.

```
Query q = new Query();
q.setClass(Persona.class);
q.setPredicate(new Predicate(
    new And(
       new Equals(new Field("nombre"), "Juan"),
       new GreaterThan(new Field("edad"), 17) )
    )
);
q.Execute();
```

- Usar lenguaje **OQL (Object Query Language**, algo así como un SQL adaptado para el uso con objetos), que permite construcciones mucho más sencillas, como la que se muestra a continuación, pero con el problema de que los datos que reciba Java (o el lenguaje orientado a objetos que se esté empleando) deberían ser de tipos conocidos de antemano.

```
SELECT p FROM Persona
WHERE p.Nombre = "Juan" AND p.Edad > 17
```

En Db4o existe una alternativa intermedia, ni tan complicada como el uso de varios predicados (que existen en Db4o pero reciben el nombre de API SODA - Simple Object Database Access) ni tan sencilla como OQL. Se llama "**Native Query**" (consultas nativas) y permite usar un único predicado, que contendrá una función "**match**" que devuelva true si el objeto que se está analizando cumple los criterios deseados, como en este ejemplo.

```
import Clases.Persona;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.query.Predicate;
import java.util.List;

public class EjemploDb4o5 {
  public static void main(String[] args) throws Exception {
    ObjectContainer db = null;
    try {
      db=Db4o.openFile("./datos/personas.db4o");

      // Buscamos las personas de más de 17 años
      List<Persona> adultos = db.query(new Predicate<Persona>() {
          @Override
          public boolean match(Persona candidato) {
            return candidato.getEdad() > 17;
          }
      });

      for (Persona adulto : adultos)
        System.out.println("Encontrado " + adulto.getNombre() + " " +
            adulto.getApellidos() + ": " + adulto.getEdad());

    } finally {
      if (db != null) db.close();
    }
  }
}
```